Docker High Performance

Docker is a portable container format that allows you to run code anywhere from your desktop to the cloud. The workflow around Docker makes development, testing, and deployment much easier and faster. However, it’s essential that you know best practices and most of the techniques for optimization so that Docker can help you deploy your application most effectively.

This comprehensive guide will improve your Docker workflows and ensure your application’s production environment runs smoothly. It starts with a short refresher on working with Docker, and then you will learn how to take this basic knowledge to the next level by optimizing your Docker infrastructure and applications at scale. At the end of the book, we will put the concepts and everything you have learned about Docker’s features into practice by rolling out supplementary monitoring and troubleshooting instrumentation to your infrastructure. All of these things will ensure your application succeeds in using Docker.

Who this book is written for

If you are a software developer with a good understanding of managing Docker services and the Linux filesystem, and are looking for ways to optimize your working with Docker containers, then this is the book for you.

What you will learn from this book

- Tune your Dockerfiles and optimize the performance and size of your Docker containers
- Configure and tune your Docker hosts at scale with Chef
- Deploy containers without downtime using load balancers
- Listen to your Docker container and host logs with an ELK stack
- Monitor the performance of your Docker containers with collectd
- Benchmark the performance of your web application containers with Apache JMeter
- Troubleshoot and diagnose containers using standard Linux diagnostic tools
- Prepare for production with the most effective DevOps practices

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 'Preparing Docker Hosts'
- A synopsis of the book’s content
- More information on *Docker High Performance*
Allan Espinosa is a DevOps practitioner living in Tokyo. He is an active open source contributor to various distributed systems tools, such as Docker and Chef. Allan maintains several Docker images for popular open source software that were popular even before their official release from the upstream open source groups.

In his career, Allan has worked on large distributed systems containing hundreds to thousands of servers in production. He has built scalable applications on various platforms ranging from large supercomputing centers in the U.S. to production enterprise systems in Japan.

Allan can be contacted through his Twitter handle @AllanEspinosa. His personal website at http://aespinosa.github.io contains several blog posts on Docker and distributed systems in general.
Docker is a great tool to build and deploy our applications. Its portable container format allows us to run code anywhere, from our developer workstations to popular cloud computing providers. The workflow around Docker makes development, testing, and deployment easier and faster. However, this is very important to Docker’s internals and continuously improving best practices to realize its full potential.

What this book covers

Engineers that have a basic understanding of Docker can read the book sequentially, chapter by chapter. Tech leads who have an advanced understanding of Docker or have deployed applications in production before can go ahead and read Chapter 8, *Onto Production*, first to understand how Docker can fit in your existing applications. The following is a list of topics covered in this book:

*Chapter 1, Preparing Docker Hosts*, gives a quick refresher on setting up and running Docker. It documents the setup that you will be using throughout the book.

*Chapter 2, Optimizing Docker Images*, shows why it is important to tune your Docker images. A few tuning tips will be shown to improve the deployability and performance of our Docker containers.

*Chapter 3, Automating Docker Deployments with Chef*, shows how to automate the provisioning and setup of Docker hosts. It will discuss the importance of investing in automation and how it facilitates a scalable way of deploying your Docker containers.

*Chapter 4, Monitoring Docker Hosts and Containers*, gives a walk-through of setting up a monitoring system with Graphite and logging systems with an Elasticsearch-Logstash-Kibana (ELK) stack.
Chapter 5, *Benchmarking*, is a tutorial on how to use Apache JMeter to create workloads to benchmark the performance of your Docker containers. The chapter reviews the monitoring system you set up in Chapter 4, *Monitoring Docker Hosts and Containers*, to analyze some Docker application benchmark results, such as response time and throughput.

Chapter 6, *Load Balancing*, shows you how to configure and deploy an Nginx-based load balancer Docker container. The chapter also gives a tutorial on how to use the load balancer you set up to scale out the performance and deployability of our Docker applications.

Chapter 7, *Troubleshooting Containers*, illustrates how common debugging tools in a typical Linux system can be used to troubleshoot your Docker containers. They describe how each tool works and how it can read the diagnostics coming from your running Docker containers.

Chapter 8, *Onto Production*, synthesizes all the performance optimizations you did in the previous chapter and relates what it means to operate any web application in production with Docker.
Preparing Docker Hosts

Docker allows us to deliver applications to our customers faster. It simplifies the workflows needed to get code from development to production by enabling us to easily create and launch Docker containers. This chapter will be a quick refresher on how to get our environment ready to run a Docker-based development and operations workflow by:

- Preparing a Docker host
- Working with Docker images
- Running Docker containers

Most parts of this chapter are concepts that we are already familiar with and are readily available on the Docker documentation website. This chapter shows selected commands and interactions with the Docker host that will be used in the succeeding chapters.

Preparing a Docker host

It is assumed that we are already familiar with how to set up a Docker host. For most of the chapters of this book, we will run our examples against the following environment, unless explicitly mentioned otherwise:

- Operating system—Debian 8.2 Jessie
- Docker version—1.10.0

The following command displays the operating system and Docker version:

```bash
$ ssh dockerhost
dockerhost$ lsb_release -a
No LSB modules are available.
Distributor ID: Debian
Description:   Debian GNU/Linux 8.2 (jessie)
```
Preparing Docker Hosts

Release:        8.2
Codename:       jessie
dockerhost$ docker version

Client:
  Version:     1.10.0
  API version: 1.21
  Go version:  go1.4.2
  Git commit:  a34a1d5
  Built:       Fri Nov 20 12:59:02 UTC 2015
  OS/Arch:     linux/amd64

Server:
  Version:     1.10.0
  API version: 1.21
  Go version:  go1.4.2
  Git commit:  a34a1d5
  Built:       Fri Nov 20 12:59:02 UTC 2015
  OS/Arch:     linux/amd64

If we haven't set up our Docker environment yet, we can follow the instructions on
the Docker website found at https://docs.docker.com/installation/debian to
prepare our Docker host.

Working with Docker images

Docker images are artifacts that contain our application and other supporting
components to help run it, such as the base operating system, runtime and
development libraries, and so on. They get deployed and downloaded into Docker
hosts in order to run our applications as Docker containers. This section will cover
the following Docker commands to work with Docker images:

- docker build
- docker images
Most of the material in this section is readily available on the Docker documentation website at https://docs.docker.com/userguide/dockerimages.

### Building Docker images

We will use the Dockerfile of training/webapp from the Docker Education Team to build a Docker image. The next few steps will show us how to build this web application:

1. To begin, we will clone the Git repository of webapp, which is available at https://github.com/docker-training/webapp via the following command:

   ```bash
dockerhost$ git clone https://github.com/docker-training/webapp.
   git training-webapp
   Cloning into 'training-webapp'...
   remote: Counting objects: 45, done.
   remote: Total 45 (delta 0), reused 0 (de..., pack-reused 45
   Unpacking objects: 100% (45/45), done.
   Checking connectivity... done.
   
   1. Then, let's build the Docker image with the `docker build` command by executing the following:

   ```bash
dockerhost$ cd training-webapp
dockerhost$ docker build -t hubuser/webapp .
   Sending build context to Docker daemon 121.3 kB
   
   Step 0 : FROM ubuntu:14.04
   Repository ubuntu already being ... another client. Waiting.  
   ---> 6d4946999d4f
   Step 1 : MAINTAINER Docker Education Team <education@docker.com>
   ---> Running in 0fd24c915568
   ---> e835d0c77b04
   Removing intermediate container 0fd24c915568
   Step 2 : RUN apt-get update
   ---> Running in 45b654e66939
   Ign http://archive.ubuntu.com trusty InRelease
   ...
   Removing intermediate container c08be35b1529
   ```

Preparing Docker Hosts

Step 9: CMD python app.py
  ---> Running in 48632c5fa300
  ---> 55850135bada
Removing intermediate container 48632c5fa300
Successfully built 55850135bada

The -t flag is used to tag the image as hubuser/webapp. Tagging containers as <username>/<imagename> is an important convention to be able to push our Docker images in the later section. More details on the docker build command can be found at https://docs.docker.com/reference/commandline/build or by running docker build --help.

3. Finally, let's confirm that the image is already available in our Docker host with the docker images command:

```
dockerhost$ docker images
REPOSITORY      TAG      IMAGE ID  CREATED        VIRTUAL SIZE
hubuser/webapp  latest   55850135  5 minutes ago  360 MB
ubuntu          14.04    6d494699  3 weeks ago    188.3 MB
```

Pushing Docker images to a repository

Now that we have made a Docker image, let's push it to a repository to share and deploy across other Docker hosts. The default installation of Docker pushes images to Docker Hub. Docker Hub is a publicly hosted repository of Docker, Inc., where anyone with an account can push and share their Docker images. The following steps will show us how to do this:

1. Before being able to push to Docker Hub, we will need to authenticate with the docker login command, as follows:

```
dockerhost$ docker login
Username: hubuser
Password: ********
Email: hubuser@hubemail.com
WARNING: login credentials saved in /home/hubuser/.dockercfg.
Login Succeeded
```

If we don't have a Docker Hub account yet, we can follow the instructions to sign up for an account at https://hub.docker.com/account/signup.
2. We can now push our images to Docker Hub. As mentioned in the previous section, the tag of the image identifies <username>/<imagename> in the repository. Issue the `docker push` command shown as follows in order to push our image to Docker Hub:

```
dockerhost$ docker push hubuser/webapp
The push refers to a repository [hubuser/webapp] (len: 1)
Sending image list
Pushing repository hubuser/webapp (1 tags)
428b411c28f0: Image already pushed, skipping
...
7d04572a66ec: Image successfully pushed
55850135bada: Image successfully pushed
latest: digest: sha256:b00a3d4e703b5f9571ad6a... size: 2745
```

Now that we have successfully pushed our Docker image, it will be available in Docker Hub. We can also get more information about the image we pushed in its Docker Hub page, which is similar to that shown in the following image. In this example, our Docker Hub URL is `https://hub.docker.com/r/hubuser/webapp`:

![Docker Hub Image Page](image)

More details on pushing Docker images to a repository are available at `docker push --help` and `https://docs.docker.com/reference/commandline/push`. 
Docker Hub is a good place to start hosting our Docker images. However, there are some cases where we want to host our own image repository. For example, when we want to save bandwidth when pulling images to our Docker hosts. Another reason could be that our Docker hosts inside a datacenter may have firewalled off the Internet. In Chapter 2, Optimizing Docker Images, we will discuss in greater detail how to run our own Docker registry to have an in-house repository of Docker images.

Pulling Docker images from a repository

Once our Docker images are built and pushed to a repository, such as Docker Hub, we can pull them to our Docker hosts. This workflow is useful when we first build our Docker image in our development workstation Docker host and want to deploy it to our production environment's Docker host in the cloud. This removes the need to rebuild the same image in our other Docker hosts. Pulling images can also be used to grab the existing Docker images from Docker Hub to build over our own Docker images. So, instead of cloning the Git repository as we did earlier and redoing the build in another one of our Docker hosts, we can pull it instead. The next few steps will walk us through pulling the hubuser/webapp Docker image that we just pushed earlier:

1. First, let's clean our existing Docker host to make sure that we will download the image from Docker Hub. Type the following command to make sure we have a clean start:

   
   dockerhost$ dockerhost rmi hubuser/webapp

2. Next, we can now download the image using docker pull, as follows:

   
   dockerhost$ docker pull hubuser/webapp
   latest: Pulling from hubuser/webapp
   e9e06b06e14c: Pull complete
   ...
   b37deb56df95: Pull complete
   02a8815912ca: Already exists
   Digest: sha256:06e9c1983bd6d5db5fba376ccd63bfa529e8d02f23d5
   Status: Downloaded newer image for hubuser/webapp:latest

3. Finally, let's confirm again that we have downloaded the image successfully by executing the following command:

   
   dockerhost$ docker images

   
   REPOSITORY          TAG       IMAGE ID       CREATED         VIRTUAL SIZE
   ubuntu              14.04     6d494699      3 weeks ago    188.3 MB
   hubuser/webapp      latest    2a8815ca      7 weeks ago    348.8 MB
More details on how to pull Docker images is available at docker pull --help and https://docs.docker.com/reference/commandline/pull.

Running Docker containers

Now that we have pulled or built Docker images, we can run and test them with the docker run command. This section will review selected command-line flags that we will use throughout the succeeding chapters. This section will also use the following Docker commands to get more information about the Docker containers being run inside the Docker host:

- docker ps
- docker inspect

More comprehensive details on all the command-line flags are found at docker run --help and https://docs.docker.com/reference/commandline/run.

Exposing container ports

In the training/webapp example, its Docker container is run as a web server. To have the application serve web traffic outside its container environment, Docker needs information on which port the application is bound to. Docker refers to this information as exposed ports. This section will walk us through how to expose port information when running our containers.

Going back to the training/webapp Docker image we worked on earlier, the application serves a Python Flask web application that listens to port 5000, as highlighted here in webapp/app.py:

```python
import os
from flask import Flask
app = Flask(__name__)
@app.route('/

import os
from flask import Flask
app = Flask(__name__)
@app.route('/
def hello():
    provider = str(os.environ.get('PROVIDER', 'world'))
    return 'Hello ' + provider + '!'
if __name__ == '__main__':
    # Bind to PORT if defined, otherwise default to 5000.
    port = int(os.environ.get('PORT', 5000))
    app.run(host='0.0.0.0', port=port)
```
Preparing Docker Hosts

Correspondingly, the Docker image makes the Docker host aware that the application is listening on port 5000 via the `EXPOSE` instruction in the Dockerfile, which can be described as follows:

```
FROM ubuntu:14.04
MAINTAINER Docker Education Team <education@docker.com>
RUN apt-get update
RUN DEBIAN_FRONTEND=noninteractive apt-get \
    install -y -q python-all python-pip
ADD ./webapp/requirements.txt /tmp/requirements.txt
RUN pip install -qr /tmp/requirements.txt
ADD ./webapp /opt/webapp/
WORKDIR /opt/webapp
EXPOSE 5000
CMD ["python", "app.py"]
```

Now that we have a basic idea of how Docker exposes our container's ports, follow the next few steps to run the `hubuser/webapp` container:

1. Use `docker run` with the `-d` flag to run the container as a daemon process, as follows:
   ```
dockerhost$ docker run --name ourapp -d hubuser/webapp
   ```

2. Finally, confirm that the Docker host has the container running with port 5000 exposed with `docker ps`. We can do this through the following command:
   ```
dockerhost:-/training-webapp$ docker ps
CONTAINER ID  IMAGE  ...   STATUS        PORTS    NAMES
df3e6b788fd8  hubuser...   Up 4 seconds  5000/tcp ourapp
```

In addition to the `EXPOSE` instruction, exposed ports can be overridden during runtime with the `--expose=[]` flag. For example, use the following command to have the `hubuser/webapp` application expose ports 4000-4500:

```
dockerhost$ docker run -d --expose=4000-4500 \
    --name app hubuser/webapp
```

```
dockerhost $ docker ps
CONTAINER ID   IMAGE      ...              PORTS                   NAMES
ca4dc1da26d    hubuser/webapp:latest ...  4000-4500/tcp,5000/tcp app
df3e6b788fd8   hubuser/webapp:l...         5000/tcp                ourapp
```
This ad hoc `docker run` flag is useful when debugging applications. For example, let's say our web application uses ports 4000-4500. However, we normally don't want these ranges to be available in production. We can then use `--expose=[]` to enable it temporarily to spin up a debuggable container. Further details on how to use techniques such as this to troubleshoot Docker containers will be discussed in Chapter 7, *Troubleshooting Containers*.

### Publishing container ports

Exposing only makes the port available inside the container. For the application to be served outside its Docker host, the port needs to be published. The `docker run` command uses the `-P` and `-p` flags to publish a container’s exposed ports. This section talks about how to use these two flags to publish ports on the Docker host.

#### --publish-all

The `-P` or `--publish-all` flag publishes all the exposed ports of a container to random high ports in the Docker host port within the ephemeral port range defined in `/proc/sys/net/ipv4/ip_local_port_range`. The next few steps will go back to the `hubuser/webapp` Docker image that we were working on to explore publishing exposed ports:

1. First, type the following command to run a container publishing all the exposed ports:
   ```sh
dockerhost$ docker run -P -d --name exposed hubuser/webapp
   ```

2. Next, let's confirm that the Docker host publishes port 32771 to forward traffic to the Docker container's exposed port 5000. Type the `docker ps` command as follows to perform this verification:
   ```sh
dockerhost$ docker ps
   CONTAINER ID IMAGE ... PORTS                     NAMES
   508cf1fb3e5  hubuser/webapp:latest ... 0.0.0.0:32771->5000/tcp   exposed
   ```

3. We can also verify that the allocated port 32771 is within the configured ephemeral port range of our Docker host:
   ```sh
dockerhost$ cat /proc/sys/net/ipv4/ip_local_port_range
   32768   61000
   ```

4. In addition, we can confirm that our Docker host is listening on the allocated port 32771 as well via the following command:
   ```sh
dockerhost$ ss -lt 'sport = ::32771'
   State  Recv-Q Send-Q  Local Address:Port Peer Address:Port
   LISTEN 0      128 :::32771          :::*
5. Finally, we can validate that the Docker host's port 32771 is indeed mapped to the running Docker container by confirming that it is the training/webapp Python application responding by making an actual HTTP request. Run the following command to confirm:

$ curl http://dockerhost:32771
Hello world!

--publish

The -p or --publish flag publishes container ports to the Docker host. If the container port is not yet exposed, the said container will also be exposed. According to the documentation, the -p flag can take the following formats to publish container ports:

- containerPort
- hostPort:containerPort
- ip:containerPort
- ip:hostPort:containerPort

By specifying the hostPort, we can specify which port in the Docker host the container port should be mapped to instead of being assigned a random ephemeral port. By specifying ip, we can restrict the interfaces that the Docker host will accept connections from to relay the packets to the mapped Docker container's exposed port. Going back to the hubuser/webapp example, the following is the command to map the Python application's exposed port 5000 to our Docker host's port 80 on the loopback interface:

$ ssh dockerhost
dockerhost$ docker run -d -p 127.0.0.1:80:5000 training/webapp
dockerhost$ curl http://localhost
Hello world!
dockerhost$ exit
logout
Connection to dockerhost closed.
$ curl http://dockerhost
curl: (7) Failed connect to dockerhost:80; Connection refused

With the preceding invocation of docker run, the Docker host can only serve HTTP requests in the application from http://localhost.
Linking containers

The published ports described in the previous section also allow containers to talk to each other by connecting to the published Docker host ports. Another way to directly connect containers with each other is establishing container links. Linked containers allow a source container to send information to the destination containers. It enables the communicating containers to discover each other in a secure manner.

More details about linked containers can be found on the Docker documentation site at https://docs.docker.com/userguide/dockerlinks.

In this section, we will work with the --link flag to connect containers securely. The next few steps give us an example of how to work with linked containers:

1. As preparation, make sure that our hubuser/webapp container runs with only the exposed ports. We will create a container called source that will serve as our source container. Type the following command to recreate this container:

   dockerhost$ docker run --name source -d hubuser/webapp

2. Next, we will create a destination container. We will use --link <source>:<alias> to create a link from the source container named source to an alias called webapp. Type the following command to create this link to our destination container:

   dockerhost$ docker run -d --link source:webapp --name destination busybox /bin/ping webapp

3. Let’s now confirm that the link was made by inspecting the newly created destination container called destination. Execute the following command:

   dockerhost$ docker inspect -f "{{ .HostConfig.Links }}" destination

   [/source:/destination/webapp]

What happened during the linking process was that the Docker host created a secure tunnel between the two containers. We can confirm this tunnel in the Docker host’s iptables, as follows:

   dockerhost$ docker inspect -f "{{ .NetworkSettings.IPAddress }}" source

   172.17.0.15

   dockerhost$ docker inspect -f "{{ .NetworkSettings.IPAddress }}" destination

   172.17.0.15
Preparing Docker Hosts

172.17.0.28
dockerhost$ iptables -L DOCKER
Chain DOCKER (1 references)

<table>
<thead>
<tr>
<th>target</th>
<th>prot</th>
<th>opt</th>
<th>source</th>
<th>destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCEPT</td>
<td>tcp</td>
<td>--</td>
<td>172.17.0.28</td>
<td>172.17.0.15</td>
</tr>
<tr>
<td>ACCEPT</td>
<td>tcp</td>
<td>--</td>
<td>172.17.0.15</td>
<td>172.17.0.28</td>
</tr>
</tbody>
</table>

In the preceding iptables, the Docker host allowed the destination container called destination (172.17.0.28) to accept outbound connections to port 5000 of the source container called source (172.17.0.15). The second iptable's entry allows the container called source to receive connections to its port 5000 from the container called destination.

In addition to the secure connections established by the Docker host between containers, the Docker host also exposes information about the source container to the destination container through the following:

- Environment variables
- Entries in /etc/hosts

These two sources of information will be further explored in the next section as an example use case of working with interactive containers.

Interactive containers

By specifying the -i flag, we can specify that a container running in the foreground is attached to the standard input stream. By combining it with the -t flag, a pseudoterminal is also allocated to our container. With this, we can use our Docker container as an interactive process, similar to normal shells. This feature is useful when we want to debug and inspect what is happening inside our Docker containers. Continuing from the previous section, we can debug what happens when containers are linked through the following steps:

1. To prepare, type the following command to establish an interactive container session linking to the container called source that we ran earlier:
   dockerhost$ docker run -i -t --link source:webapp \
   --name interactive_container \ 
   busybox /bin/sh
   /
   #
2. Next, let's first explore the environment variables that are exposed to the interactive destination container via the following command:

```
/ # env | grep WEBAPP
```

```
WEBAPP_NAME=/interactive_container/webapp
WEBAPP_PORT_5000_TCP_ADDR=172.17.0.15
WEBAPP_PORT_5000_TCP_PORT=5000
WEBAPP_PORT_5000_TCP_PROTO=tcp
WEBAPP_PORT_5000_TCP=tcp://172.17.0.15:5000
WEBAPP_PORT=tcp://172.17.0.15:5000
```

In general, the following environment variables are set in linked containers:

- `<alias>_NAME=/container_name/alias_name` for each source container
- `<alias>_PORT_<port>_<protocol>` shows the URL of each exposed port. It also serves as a unique prefix expanding to the following more environment variables:
  - `<prefix>_ADDR` contains the IP address of the source container
  - `<prefix>_PORT` shows the exposed port's number
  - `<prefix>_PROTO` describes the protocol of the exposed port which is either TCP or UDP
- `<alias>_PORT` shows the source container's first exposed port

3. The second container discovery feature in linked containers is an updated `/etc/hosts` file. The alias of the `webapp` linked container is mapped to the IP address of the source container. The name of the source container is also mapped to the same IP address. The following snippet is the content of the `/etc/hosts` file inside our interactive container session, and it contains this mapping:

```
172.17.0.29     d4509e3da954
127.0.0.1       localhost
::1     localhost ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
172.17.0.15     webapp 85173b8686fc source
```
Preparing Docker Hosts

4. Finally, we can use the alias to connect to our source container. In the following example, we will connect to the web application running in our source container by making an HTTP request to its alias, \texttt{webapp}:

\texttt{/ # nc webapp 5000}

\texttt{GET /}

\texttt{Hello world!}

\texttt{/ #}

Interactive containers can be used to build containers as well, together with \texttt{docker commit}. However, this is a tedious process, and this development process doesn't scale beyond a single developer. Use \texttt{docker build} instead and manage our \texttt{Dockerfile} in version control.

Summary

Hopefully by this time, we are refamiliarized with most of the commands that will be used throughout the book. We prepared a Docker host to be able to interact with Docker containers. We then built, downloaded, and uploaded various Docker images to develop and deploy containers to our development and production Docker hosts alike. Finally, we ran Docker containers from built or downloaded Docker images. In addition, we established some basic skills of how to communicate and interact with running containers by learning about how Docker containers are run.

In the next chapter, you'll learn how to optimize our Docker images. So, let's dive right in!
Where to buy this book

You can buy Docker High Performance from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.